Programming Idioms pt 2 -- Design Patterns

Tennessee Leeuwenburg

I'll admit it – I live under a rock. I hadn't heard about Design Patterns until a couple of months ago. So sue me. That said, I'll bet I'm not the only one. So what's a Design Pattern?

Simple. It's boilerplate design. The idea seems simple enough – many problems can be expressed in common ways. If that's true, then there should exist some small set of program structures which are useful for many, many problems.

Identifying these and documenting them is what is involved in creating a Design Pattern. Armed with a library of Design Patterns, solving a complex problem can be simplified by expressing it in terms of a problem that you *already* know how to solve – at least more or less.

The seminal text in this area is *Design Patterns: Elements of Reusable Object-Oriented Software*[9], often referred to as the *GoF* or *Gang of Four*, partly because it has four authors and partly because it is a much shorter title.

One of the first things that occurred to me when I was thinking about this topic was *MapReduce*[10], an algorithm used by Google in distributed processing.

Many problems can be expressed in terms of operations which also happen to be easily run in parallel. If a problem can be expressed in terms of a Map() and then Reduce() function, then it is easy to make use of distributed processing.

While MapReduce is not one of the patterns addressed in *GoF*, many that are may already be familiar to many programmers. These include Factories, Singleton classes, Adapters, Iterators and others.

They have all been built many times over because they are a sensible way to deconstruct a program design. Design Patterns are an attempt to bring additional tools born from experience to bear on problem analysis and code design.

There are further similarities with, for example, Agent-Oriented architectures and other such design systems. If they have anything in common, it is because there is an overriding truth that a complex problem must often be broken into component parts before a solution may be found.

A programming idiom is the lowest-level expression of this truth. As discussed in part one of this series, a programming idiom is a practise adopted to help make the meaning of code clearer.

It is perhaps reasonable to say then that a Design Pattern is a practise adopted to help make the medium-scale structure of code clearer.

Design Patterns for Python

Many Design Patterns appear to be closely linked to particular Object-Oriented models which do not apply to Python as neatly as to other languages. For example, Python's dynamic duck typing obviates some of the patterns which relate to issues caused by static typing, for example the Abstract Factory and Builder Patterns.

For specific information on how to use design patterns in your own code, “Design Patterns in Python”[11] is a good start. However, I will briefly list the Design Patterns which appear to me to be most relevant to Python. These patterns are, in no particular order: Prototype; Singleton

9 http://www.amazon.com/Design-Patterns-Object-Oriented-Addison-Wesley-Professional/dp/0201633612
10 http://labs.google.com/papers/mapreduce.html
11 http://www.python.org/workshops/1997-10/proceedings/savikko.html

(optionally with Borg style); Composite; Facade; Flyweight; Proxy (*looks Evil, Ed*); Chain of Responsibility; Command; Iterator; Interpreter; Mediator; Memento; Observer (or Listener) and Strategy.

Having read up on each of these (see GoF or else Wikipedia[12] has an entry for each, see “Gang of Four”), I made a sub-list of patterns that I have used myself: Strategy; Observer (or Listener); Iterator; Chain of Responsibility; Facade; Composite and Singleton.

However, I have rarely used these patterns as part of an overarching design strategy. Indeed, it is still unclear how these patterns may be best integrated into system design for significant components or applications. They appear more applicable to small-scale problems.

It is not clear, for example, how the Iterator pattern applies to a larger task – implementing a Customer Relations Manager for example. Only once that larger system is broken down to include a component which processes each customer does an Iterator become useful in solving the problem.

Design Patterns, then, are a higher level of abstraction than an algorithm, but a lower level of abstraction than system design.

When to use Design Patterns

The question then is, 'When should a Design Pattern be used?' If not during system design (too low-level) and not line-by-line (too high-level), then when?

There are no hard and fast rules. Little academic research has been done on the topic, and there are no de-facto standards here. As such, if you see an opportunity to make use of a Design Pattern, doing so may well save you much time spent in the design phase.

Typically, the right time is when working on an individual component of a larger system. For example, supposing the task at hand were to build an Internet chat room. This might require a central server to which multiple clients could connect, some kind of user interface for interacting with the system, and messaging processing logic on both sides. This is the system design level.

Let's dive into the server logic. The server will need to manage client connections and handle the relevant message-passing. A simple server could be structured around a simple main loop (Iterator pattern):

```
class ChatServer:

    def _mainLoop(self):
        outgoing = ''

        # Get messages coming in from each client
        for client in clients:
            incoming = client.getMessages()
            outgoing = outgoing + '\n'.join(incoming)

        # Push the message stack back to each client
        for client in clients:
            client.recieveMessages(outgoing)
```

Here we see the use of an iterator which is central to the Python language. The ability of *for* loops to process interated information is very useful. It allows iteration over lists, sequences or even user-defined functions.

The Iterator pattern can also be used from the other side, so to speak, in writing code which

12 www.wikipedia.org

generates the items to be iterated over. It could be used, for example, as the basis of an event-based system. Writing an Iterator which *yields* events would allow a simple *for* loop to control program execution.

It can be seen that there are uses for Design Patterns just below the level of system design – when building components identified during this earlier phase. This process of taking a design requirement, and looking for a Design Pattern which might be able to meet that requirement, is not just an exercise for its own sake. Rather, it is a logical short-cut. Implementing a problem in an already familiar way will make the job quicker and simpler. Should the code ever be passed to another programmer, they will be able to understand the execution flow that much more easily.

To use another analogy, using Design Patterns liberally will encourage the use of common structures in various system components, indeed across programs and even across programming languages and systems. While a Java implementation may not be syntactically the same as a Python implementation, the underlying structure of the code will be similar.